A service based estimation method for MPSoC performance modelling

Tranberg-Hansen, Anders Sejer; Madsen, Jan; Jensen, Bjørn Sand

Published in:
2008 International Symposium on Industrial Embedded Systems

Link to article, DOI:
10.1109/SIES.2008.4577679

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

A Service Based Estimation Method for MPSoC Performance Modelling

Anders Sejer Tranberg-Hansen and Jan Madsen
Technical University of Denmark
DK-2800 Kgs. Lyngby, Denmark
e-mail: {asth,jan}@imm.dtu.dk

Bjørn Sand Jensen
Bang & Olufsen ICEpower a/s
DK-2800 Kgs. Lyngby, Denmark
http://www.icepower.bang-olufsen.com/

***Abstract*— This paper presents an abstract service based estimation method for MPSoC performance modelling which allows fast, cycle accurate design space exploration of complex architectures including multi processor configurations at a very early stage in the design phase. The modelling method uses a service oriented model of computation based on Hierarchical Colored Petri Nets and allows the modelling of both software and hardware in one unified model.**

To illustrate the potential of the method, a small MPSoC system, developed at *Bang & Olufsen ICEpower a/s*, is modelled and performance estimates are produced for various configurations of the system in order to explore the best possible implementation.

I. Introduction

In order to address the challenges associated with the increasing architectural design complexity seen in modern VLSI systems, methods for early and accurate design space exploration are needed, allowing evaluation and selection of the best possible configuration of a given system without compromising the overall time-to-market of the system.

This paper presents an abstract hardware/software modelling and performance estimation method for performing design space exploration of MPSoC systems. The method allows MPSoC designers to model and analyze hardware and software components of a system and their interaction, using a unified service-based modelling approach at a very early stage in the design phase. The method, outlined in figure 1, is simulation-based and uses a service oriented model of computation which is based on a modified version of Hierarchical Colored Petri Nets (*HCPN*) [1].

Performance estimation is done at system level using *system models*. A system model of an MPSoC system is constructed by mapping the contents of one or more *application models* onto the processing elements of a *platform model*. The platform model of the target architecture is composed of one or more *component models*, each modelling a hardware component of the target architecture e.g. a processing element, a memory element, a functional unit, an inter-connection structure, etc. Component models are implemented as *service models* [2], modelling the behaviour of the actual hardware component through the availability of a set of services. An application model, thus, represents the behaviour of an application by requesting a sequence of the services offered by the service model of the processing element onto which the application is mapped.

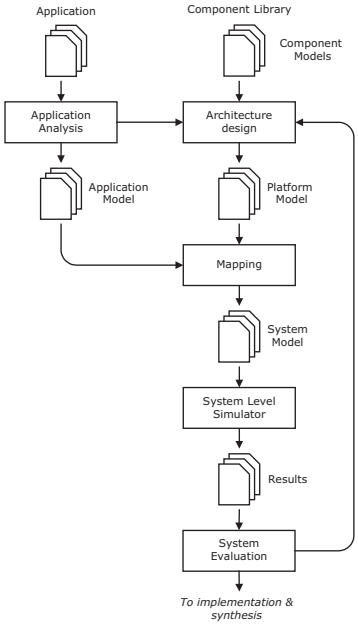

Fig. 1. Overview of the proposed modelling and performance estimation method.

The simulation of a system model makes it possible to produce detailed information regarding the runtime properties of the applications and of the specified platform, e.g. execution profiles, resource utilization, stalls and their causes, memory usage, communication channels, etc. and so can direct the designer to the elements of the system which ought to receive special attention.

The advantages of the modelling method presented in this paper are the flexibility, the refinement possibilities of models, and the high level of accuracy which is obtainable. Service models have well defined interfaces and together with the hierarchical composition properties inherited from HCPNs it is possible to combine models described at different levels of abstraction into one model, allowing a gradual refinement of the details of a model, or interchanging models, in order to investigate different implementations.

978-1-4244-1995-1/08/$25.00 ©2008 IEEE.

II. Related Work

Several MPSoC modelling and performance estimation methods and frameworks for performing design space exploration exist e.g. [3], [4], [5], [6], [7].

MPSoC simulation methods supporting mixed abstraction level co-simulations and a step-by-step refinement of models, such as the method presented in this paper, have been seen recently, e.g. Metropolis [3] which aims to provide a framework in which models can be described at multiple levels of abstraction and gradually refined. It allows multiple models of computations to co-exist within the framework by defining common communication semantics which allow the different models to communicate.

Trace driven simulation methods, in which applications are modelled by the generation of event traces, is found, e.g., in the Artemis [6] sub-project Sesame [8]. Each event represents some kind of behaviour found in the application, and the resources required to execute the behaviour is modelled when the event is evaluated by an architecture model. The method presented in this paper represents applications by a sequence of service requests very similar to event traces and thus, to some extent, the method can be categorized as a trace driven method.

HCPNs, which are used as the model of computation in the current method, are extremely powerful, flexible and mathematically well defined. Thus, they are used in a number of applications [1]. However, the use of HCPNs for detailed co-simulation is rather limited in both the academic and industrial worlds. This is due to the complexity required of capturing the detailed behaviour of e.g. a processor pipeline in HCPNs.

[9] presents an approach for performance estimation of pipelined processors and cycle accurate instruction set simulator generation. However, their goal is to develop fast instruction set simulators and they focus on pipelined processors only.

Service models are presented as part of a modelling and performance estimation framework in [2]. The basic idea of service models is very intuitive from a hardware designer's point of view and it is a very appealing method of modelling hardware/software co-design problems. However, service models are only intended for modelling a single top level hardware component and, thus, cannot be used for modelling MPSoC systems. This issue is addressed in the method presented in this paper and the basic service model approach is extended, making service models usable in the current context.

III. System Modelling and Performance Estimation

This section will introduce the elements of which the method is composed and explain how performance estimation is carried out allowing MPSoC designers to perform early and accurate design space exploration.

A. Service models

A Service model is an abstract model of a hardware component modelling the behaviour of the component by a set of services. Depending on the level of abstraction used to describe the service model, a service can represent anything from the execution of a task or a function to arithmetic operations or actual instructions. The service model defines which services are provided, how their behaviour is implemented and how long time a service needs to execute.

Service models are composed of a *service model interface* and a *service model implementation*. The service model interface provides a uniform interface which all service models must implement. The unified interface simplifies the control of the service model during simulation. The detailed behaviour and function of the hardware component being modelled is implemented by the service model implementation.

Before conceptually introducing service models, it should be noted that service models are implemented as a special type of HCPNs. Without going into the details of HCPNs, the major difference is that in order to model hardware, a special type of execution semantics is needed.

In the current implementation only synchronous hardware consisting of a single clock domain can be modelled. However, the type of hardware that can be modelled is determined by the execution semantics employed. Support for multiple clock domains, as well as asynchronous components, can be added simply by changing the execution semantics.

In order to model the occurrence of a global clock event in HCPN terminology, the maximum step is always chosen to occur. The maximum step corresponds to the largest obtainable step size, i.e. the maximum possible number of enabled transitions is contained in the step.

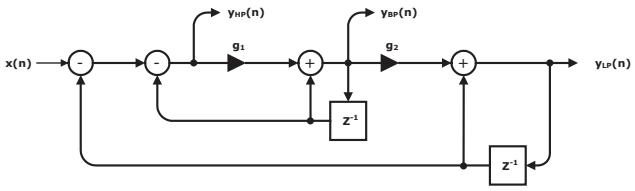

Fig. 2. Structure of a SVF filter. $y_{HP}(n)$,$y_{BP}(n)$ and $y_{LP}(n)$ denotes high pass, band pass and low pass outputs respectively.

The basic concepts of service models will be introduced by considering a small application-specific processor developed at *Bang & Olufsen ICEpower a/s*, referred to as the *SVF processor*. The SVF processor is a synchronous fixed point processor having an instruction set designed to be optimized for running applications composed of state variable filter structures (*SVF*) [10]. SVF filter structures, illustrated in figure 2, are ideal for systems where simple relations between filter coefficients and pole placement are desired, such as in embedded parameterized systems.

The instruction set of the processor consists of eighteen instructions only and has a very shallow pipeline as illustrated in figure 3. All instructions are executed during three clock cycles, except for the dedicated SVF instruction which is implemented using four clock cycles due to the complexity of the instruction. The SVF processor also implements a simple CSP-like protocol used for inter-processor communication [11]. In this protocol, send and received commands are used to

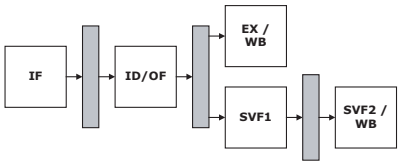

Fig. 3. Block diagram of the SVF-processor pipeline. IF = Instruction fetch stage, ID/OF = Instruction decode / Operand fetch stage, EX/WB = Execute and write back stage, SVF1 = First half of state variable filter calculation and finally SVF2/WB = Second half of state variable filter calculation and write back of results.

exchange data between processors. Receive commands are blocking i.e. the pipeline stalls until data from the specified source has been received. Send commands are non-blocking as long as the transmit buffer of the interface is not full.

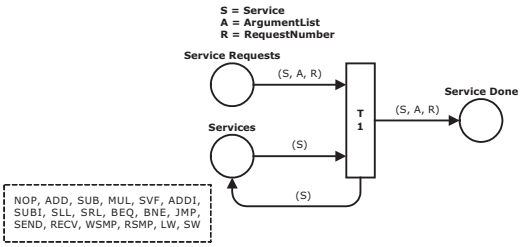

Fig. 4. Service model interface of the SVF processor. Note the service tokens representing the instruction set of the processor which are initially placed in the **Services** place.

1) Service model interface of the SVF processor: The service model interface, implemented by all service models, defines three places: **Service Requests**, **Services** and **Service Done**, a single transition **T1** as well as the four connections i.e. arcs shown in figure 4.

The instructions of the SVF processor is represented by a number of services and initially, one token for each service provided is put in the place **Services**, e.g. one token representing addition **ADD**, another token representing multiplication **MUL** and so on. The services are essentially the only elements of the service model interface which differs between the service model interface of different service models.

During the simulation of a service model, a service is requested by placing a *service request* token in the place **Service Requests**. A service request specifies the requested service, an optional empty list of arguments and a unique request number used to identify the service request used by the simulation engine e.g. to annotate the execution time of the service request. The argument list can be used to provide input arguments to the implementation of a service or to allow the modelling of data operand dependencies by letting the arguments specify one or more data operands that must be present before the service can be executed.

If the requested service is available in the **Services** place, the transition **T1** becomes enabled and is allowed to fire in the next simulation cycle: i.e. the transition is enabled. During a simulation cycle, all concurrently enabled transitions are fired corresponding to the modelling of a global clock event.

When the transition **T1** fires, the service request token from the **Service Requests** place and the corresponding service token from the **Services** place is consumed. The firing of the transition produces a new service token - of the type that was just consumed - in the **Services** place indicating that the model is ready for executing the same service again in the next simulation cycle. Furthermore, a service request token is produced in the **Service Done** place, having the same arguments as the service requests consumed including the unique request number which identifies the service request. The arrival of the service request in the **Service Done** place indicates the completion of the service request.

Thus, at this level of abstraction, all instructions complete in one cycle which does not resemble the behaviour of the actual SVF processor pipeline. It is therefore necessary to refine the model by substituting the transition of the service model interface with the service model implementation of the SVF processor described in the following section.

2) Service model implementation: In order to model the behaviour, i.e. the function and the correct execution time, of the instructions of the SVF processor, the pipeline of the processor is modelled by a number of places, arcs and transitions as shown in figure 5. The places containing *I*, *O* or *I/O* marks, represent input, output or input/outputs of the service model implementation. The figure also illustrates how conditional arc expressions are used to route the tokens to the correct places of the model, such that e.g. the SVF instructions are modelled as completing in four clock cycles as opposed to the remaining instructions which complete in three cycles. The implementation of the function of the instructions modelled is done by associating actions with the transitions of the service model implementation. The actions are executed when the transitions are allowed to fire. These, however, are not shown in the figure.

The CSP-like inter-processor communication interface implemented by the SVF processor, is modelled by the two places **RX** and **TX** respectively. The firing of transition **T2** only produces a service request token in the **TX** when executing a **SEND** service request due to the conditional arc expression, modelling that data are being placed in the transmit queue. The argument list of the **SEND** service request specifies the data being transferred as well as the sender's source address and the receiver's destination address.

Similarly, the firing of **T3** is only affected when a **RECV** service request is being executed. **T3** then requires that a service request token is available in the **RX** place having the source address specified by the argument list of the **RECV** service request as well as the correct destination address. If this is not the case, the transition will not become enabled until this condition becomes true, modelling a blocking read causing the pipeline to stall.

In order to model finite transmit and receive buffers, of the inter-processor communication interface, additional conditions can be specified associated with the transition **T2** and **T3** and the places **TX** and **RX**, preventing the transitions from being enabled if the transmit or receive buffers respectively, are full.

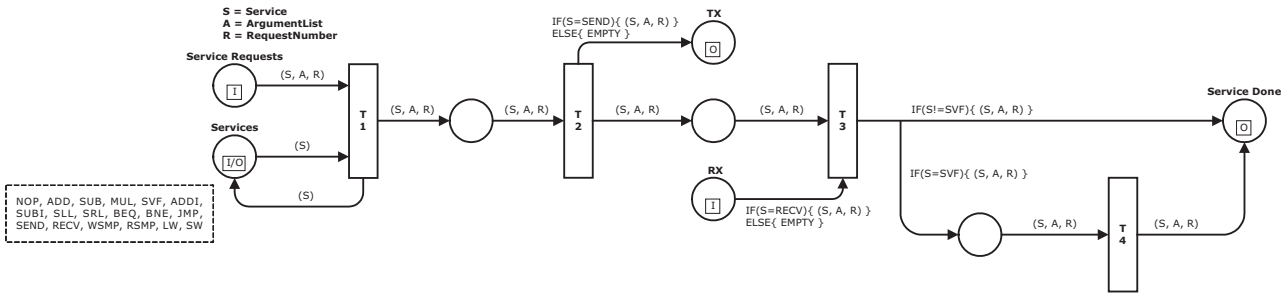

Fig. 5. Service Model implementation of the SVF-processor.

The service model implementation substitutes the **T1** transition of the service model interface and the places **Service request**, **Services** and **Service done** are connected to the corresponding places of the service model interface. Thereby, a refinement of the original model is obtained in which the instructions are carried out in the correct number of clock cycles, making the model cycle accurate. The places **RX** and **TX** are the input and output of the inter-processor communication interface being modelled, and allow the service model of the SVF processor to be connected to other service models.

Thus, to summarize, the basic structure of the service model implementation dictates the timing and concurrency properties of the SVF processor model. However, the actual behaviour of a service is implemented by associating actions with the transitions of the model. If transition actions are combined with the possibility of adding arguments to the service requests using the argument list, the actual behaviour or operation of the service can be implemented in the model allowing e.g. the implementation of an actual addition of two values, or the possibility of modelling data operand dependencies, etc. This emphasizes the great potential of the method with respect to flexibility and accuracy. It is actually possible to refine models to a level where they can be used for e.g. cycle accurate instruction set simulators if needed.

B. Platform Models

A major limitation of the service model approach presented in [2] in the context of MPSoC modelling is that it only focuses on the modelling of a single top component, e.g. a single processor, and thus, it does not define any means for modelling multi-processor architectures. In order to be able to model multi-processor architectures, and the inter-processor communication, it is necessary to define a new type of model. This is done by introducing the concept of a *platform model*. The platform model is used to model hardware architectures and is composed of one or more component models implemented as service models, each modelling a hardware component of the target architecture e.g. a processor, a functional unit, memory etc. Furthermore, the platform model specifies how the component models are interconnected. In this way the platform model can represent arbitrary target architectures including multi-processor systems.

In order to allow two or more service models to be connected to each other, the service model must define one or more input, output or input/output places. An input and an output place of two different service models can then be connected to each other forming one logical place, enabling the two service models to communicate by exchanging tokens.

There are no restrictions on how inter-component communication is modelled, thus, in principle, all types of inter-component communication methods are supported. As an example, the service model of the SVF processor, which implements a model of a CSP-like interface for inter-processor communication, is used. The model of the interface allows the SVF service models to perform inter-component communication either using point-to-point connections or using some kind of inter-connect structure such as a bus or network-on-chip.

To illustrate the use of platform models, a small MPSoC system, developed at Bang & Olufsen ICEpower a/s, is considered which consists of four SVF processor models connected via a shared bus, as illustrated in figure 6. Each processor is assigned a unique address which identifies the processor on the bus.

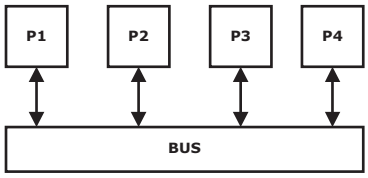

Fig. 6. Example of a simple MPSoC platform consisting of four SVF processors connected using a bus-based interconnect structure.

The bus is a shared resource implying that only one of the processors can access the bus at a time. In this simple example, a simple priority scheme is modelled which gives priority to the processor with address 0, then address 1 and so on. This is done using transition actions. There are no upper limits on the number of processors which can be connected to the bus. In this example a bus transfer is modelled to be performed during one simulation cycle, thus the service model of the bus, consists of the service model interface, shown in figure 7, only.

The **TX** output place of each SVF processor service model instance is connected to the **Service Request** place of the bus service model. Similarly, the output **Service Done** of the bus service model is connected to each of the **RX** input places of the SVF processor service model instances. Thereby,

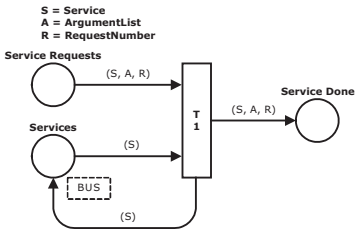

Fig. 7. The service model interface of the bus service model.

the four SVF processor service model instances are able to communicate with each other via the bus service model, modelling the interconnect structure. By changing the bus model to a model of e.g. a network-on-chip or using point-to-point connections, various inter-processor communication schemes can easily be investigated.

C. Applications

In order to perform useful design space exploration, it is vital for the modelling method that it can capture the behaviour of the applications running on the hardware architecture being modelled. In the method presented, applications are represented by an *application model* which is specified as a list of service requests, each requesting a service offered by the execution platform onto which the application has been mapped.

Service requests can be implemented at various levels of abstraction ranging from the most abstract case, as the request of a function over arithmetic operations, to instructions and actual machine code, if desired. This property gives the modelling approach a high degree of flexibility and the interpretation of the service requests is directly determined by the implementation of the service model on which the application runs. The use of service requests implies that there are no constraints on the type of applications supported, as long as an entity capable of translating the application into a list of service requests exists and that the services requested are provided by the platform which executes the application.

The method used does not infer any restrictions on the process of converting an application into an application model implemented as list of service requests. Thus, an application can either be translated by a compiler, by a simple translator capable of translating the output of an existing compiler into a list of service requests or lastly by hand. Similarly, the mapping of an application model to the processing elements of a platform model can be done manually or by a compiler depending on the implementation and the choice of the designer. This implies that the method presented can also be used for design space exploration within compiler technologies and mapping policies.

In the context of the SVF processor in focus, applications are specified at algorithm block level graphically, and mapped manually to the processors of the platform model. A custom developed compiler then compiles the application into a list of ordered service requests for each SVF processor model of the platform.

D. Performance Estimation

This section describes how performance estimation is carried out using *system models*. A system model is composed of an application mapping and a platform model.

Figure 8 shows an overview of the performance estimation method.

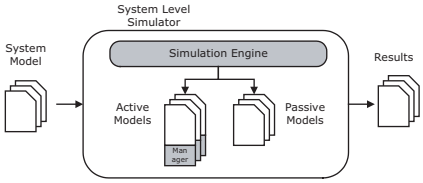

Fig. 8. Performance estimation based on the simulation of *System Models*.

The overall control of the performance estimation method is concentrated in the entity referred to as the *Simulation Engine*. However, the task of the Simulation Engine is rather simple and it basically controls the platform model by e.g. issuing initialization and cycle increment commands corresponding to the tick of a clock in a synchronous system.

The individual component models of which the platform model is composed are divided into two groups. *The first group* contains the models representing processing elements i.e. models capable of executing applications. These are referred to as *active* models and each have an associated *manager* responsible for requesting the services specified by the application mapped to the processing element being managed. The task of the manager can thus be compared to a scheduler. However, in most cases the task of the manager will be quite trivial because the list of service requests to be requested is generated during the compilation of the application prior to runtime. Furthermore, the manager is also responsible of extracting and annotating the execution time of each service request of the application when it has been executed. *The second group* of models are referred to as passive. These are the models which do not execute applications directly e.g. bus or memory models. The passive service models provide their services to other service models and thus do not have a manager associated.

The process of a simulation is as follows:

1) At initialization, the initial marking of all models are loaded to prepare the platform model for simulation and the individual managers initialize the applications that are mapped to the service model with which the manager is associated.
2) The managers of the active models checks if the **Service Requests** places of the service model interface of the service model they manage is empty, implying that a service request can be requested. If this is the case, and the manager has unrequested service requests left, the next service request is requested.
3) All transitions are now checked for enablement. The set of enabled transitions are fired consuming and producing the tokens specified by their implementation.

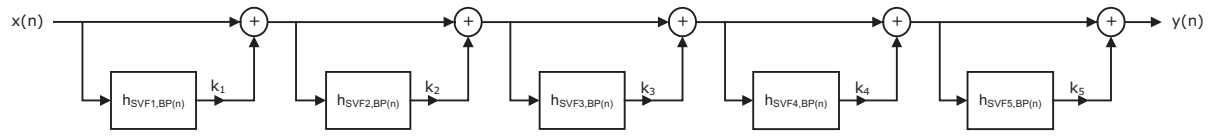

Fig. 9. Block diagram of the 5-band cascaded equalizer example application.

4) The managers remove all service requests from the **Service Done** places and annotate these with the current simulation cycle.
5) The simulation engine increments the simulation cycle and step 2 to 5 is repeated.

The managers of the active models thus collect information about the execution time of the applications running on the platform. In order to extract other runtime information, such as memory usage, resource occupancy, stalls and their causes, etc., it is possible to associate custom event loggers with the individual models. Thereby, the designers of the model can determine which properties should be logged, and the method, thus, does not limit the type of properties which can be extracted.

The performance estimation method presented is closely related to the execution of applications on the actual hardware architecture. The accuracy of the performance estimation method is therefore directly dependent on the level of details included in the models of which the platform models are composed. The modularized structure of the performance estimation method makes it quite easy to extend or refine the different elements and also implies that not only design space explorations can be performed on algorithm or architecture level, but also that e.g. the importance of compiler technology can be explored using the proposed framework.

IV. Experimental Results

In order to illustrate the use of the performance estimation method, a 5-band equalizer application from the algorithm portfolio of *Bang & Olufsen ICEpower a/s* is considered. The 5-band equalizer, illustrated in figure 9, is implemented as five cascaded band pass filters, each implemented by a state variable filter (illustrated in figure 2). The five filter blocks are identical in structure differing only by their filter coefficients.

The use of state variable filters makes the 5-band equalizer application ideal to implement using one or more of the SVF processors described in previous sections. To find the best possible platform for the application, a number of configurations are considered including various multi-core configurations. However, the exact configuration of the processor or the optimal number of processors and the inter-connection scheme are not straightforward to determine.

A number of variants of the example platform model described in section III-B have been produced ranging from a single processor platform to multi-core configurations of up to five processors (*SVFxX* - X denotes the number of SVF processors used in the configuration). The effect of being able to perform forwarding of data operands is also investigated through three different versions of the SVF processor model: *SVF#NOForward*, *SVF#Forward* and *SVF#Impl*. *SVF#NOForward* is modelled without forwarding capabilities, the *SVF#Forward* version with full forwarding capabilities, and lastly the *SVF#Impl* version is modelling the actual hardware implementation. The three types of models are all based on variants of the SVF processor model introduced in the previous sections. The differences between these are modelled by the actions associated with the transitions of the models. Furthermore, two different inter-connection schemes are investigated. The first (*SVFx1-SVFx5*) is based on direct point-to-point connections between the processors and the second (*SVFx4@BUS* and *SVFx5@BUS*) scheme is bus based.

The 5-band equalizer application, is divided into five sub-applications (one for each band) which can be mapped individually to the processors of the platform. When the applications have been mapped, they are compiled into one or more lists of service requests, one for each processor of the platform. In the configurations with multiple SVF processors, the mapping of the 5-band equalizer is done so that it is as balanced as possible e.g. for the configuration with three SVF processors, the five sub-applications are mapped as 2-2-1 etc.

TABLE I

Latency and throughput in cycles per data sample for different architectures modelling an SVF processor *without* forwarding (*SVF#NOForward*), *with* forwarding (*SVF#Forward*) and as the actual hardware implementation *SVF#Impl*.

		Latency	Throughput
SVF#NOForward	**SVFx1**	33	33
	SVFx2	35	21
	SVFx3	37	15
	SVFx4	39	15
	SVFx5	41	9
	SVFx4@BUS	45	15
	SVFx5@BUS	49	9
SVF#Forward	**SVFx1**	17	17
	SVFx2	18	11
	SVFx3	19	8
	SVFx4	20	8
	SVFx5	21	5
	SVFx4@BUS	26	8
	SVFx5@BUS	29	5
SVF#Impl	**SVFx1**	27	27
	SVFx2	28	17
	SVFx3	29	12
	SVFx4	30	12
	SVFx5	31	7
	SVFx4@BUS	36	12
	SVFx5@BUS	39	7

Table I shows the latency and throughput per sample for the different configurations and table II shows detailed information

TABLE II

UTILIZATION AND CAUSE OF STALLS DURING DESIGN EXPLORATION OF THE DIFFERENT CONFIGURATIONS OF THE PLATFORMS RUNNING THE 5-BAND EQUALIZER APPLICATION. *R-stalls* DENOTE RECEIVE STALLS EXPERIENCED WHEN A PROCESSOR NEEDS DATA FROM ANOTHER PROCESSOR, *D-stalls* DENOTE DATA DEPENDENCY STALLS CAUSED BY MISSING FORWARDING CAPABILITIES.

		SVF#NoForward			SVF#Forward			SVF#Impl		
Configuration		**Utilization**	**R-stalls**	**D-stalls**	**Utilization**	**R-stalls**	**D-stalls**	**Utilization**	**R-stalls**	**D-stalls**
SVFx1	**P1**	52.4%	0%	100%	100%	0%	0%	63.0%	0%	100%
SVFx2	**P1**	52.3%	0%	100%	100%	0%	0%	64.8%	0%	100%
	P2	37.6%	48%	52%	72.3%	100%	0%	46.7%	56%	44%
SVFx3	**P1**	53.4%	0%	100%	100%	0%	0%	66.8%	0%	100%
	P2	52.6%	3%	97%	99.3%	100%	0%	66%	3%	97%
	P3	32.7%	61%	39%	61.8%	100%	0%	40.9%	72%	28%
SVFx4	**P1**	53.4%	0%	100%	100%	0%	0%	66.8%	0%	100%
	P2	33.1%	61%	39%	62.3%	100%	0%	41.4%	72%	28%
	P3	32.8%	61%	39%	62.0%	100%	0%	41.3%	72%	28%
	P4	32.6%	62%	38%	61.8%	100%	0%	40.9%	72%	28%
SVFx5	**P1**	55.6%	0%	100%	100%	0%	0%	71.4%	0%	100%
	P2	55.1%	2%	98%	99.6%	100%	0%	71.0%	2%	98%
	P3	54.7%	4%	96%	99.2%	100%	0%	70.6%	4%	96%
	P4	54.3%	5%	95%	98.8%	100%	0%	70.2%	6%	94%
	P5	53.8%	7%	93%	98.4%	100%	0%	69.8%	8%	92%
SVFx4@BUS	**P1**	53.4%	0%	100%	100%	0%	0%	66.8%	0%	100%
	P2	33.0%	61%	39%	61.3%	100%	0%	41.3%	72%	28%
	P3	32.8%	61%	39%	62.2%	100%	0%	41.0%	72%	28%
	P4	32.3%	62%	38%	61.8%	100%	0%	40.8%	73%	27%
SVFx5@BUS	**P1**	55.6%	0%	100%	100%	0%	0%	71.4%	0%	100%
	P2	55.0%	2%	98%	99.4%	100%	0%	70.9%	3%	97%
	P3	54.5%	5%	95%	98.8%	100%	0%	70.4%	5%	95%
	P4	53.9%	7%	93%	98.2%	100%	0%	69.8%	8%	92%
	P5	53.4%	9%	91%	97.6%	100%	0%	69.2%	10%	90%

of the simulations regarding the utilization of each processor of the different architectures as well as the distribution of stalls. The utilization is defined as the cycles in which the processor is not idle or stalled, i.e. the period where actual application execution is performed. The processors of the simulations, however, are executing infinite loops and, hence, they are never idle. The processors are thus either executing application code or stalled.

As can be seen in both table I and II, not being able to forward data has a severe performance impact when executing the 5-band equalizer application. This is due to the sequential nature of the application which requires operations on a series of intermediate results in sequence.

Table I shows that using the MPSoC configurations corresponds to a high level pipelining of the equalizer application increasing the throughput at the cost of a larger latency per sample.

Independently of the number of processors of the platform, it can be seen from table II that the best utilization of the processors is obtained when a balancing of the number of applications executed per processor can be achieved, corresponding to balancing the work when pipelining a combinatorial circuit. Thus, in terms of utilization, one should either use a single processor system or a system consisting of five processors.

To obtain the optimal SVF processor in terms of utilization and throughput when executing the 5-band equalizer application, forwarding should be implemented for all data operands as can be seen from the results of *SVF#Forward* in table II. However, this is not possible unless the SVF instruction can be executed in a single clock cycle because the results otherwise simply will not be valid. The solution employed in the currently implemented SVF processor, modelled by *SVF#Impl*, provides forwarding of all general purpose register operands but not the SVF operands.

The performance estimation results also show that the penalty of using bus based interconnects instead of point-to-point connections only corresponds to the amount of time to negotiate access with the bus arbiter, and thus is a promising choice because a bus based platform provides more flexibility, if the application is to be changed after the manufacturing of the platform. The reason for the limited penalty of using a bus based interconnect vs. point-to-point connections lies in the sequential nature of the equalizer application which means that the data transfer between the processors is timed in accordance with each other in the current application and hence no congestion is seen.

A. Functional Verification

In order to functionally verify the behaviour of the detailed SVF processor model (*SVF#Impl*), the platform model consisting of a single SVF processor is used and the 5-band equalizer application is mapped to the processor. Two versions of the *SVF#Impl*, are used, one implemented using *double* data types and another implemented using the 24 bit fixed point data representation found in the actual hardware implementation.

As input to the 5-band equalizer application a multi-tone signal (20Hz-20kHz), consisting of 1764 samples, is used.

As a reference for the simulation output produced by the simulation of the constructed system model, an implementation of the 5-band equalizer application has been made in

TABLE III

MEAN AND VARIANCE OF THE DIFFERENCE BETWEEN THE SIMULATIONS AND THE MATLAB REFERENCE IMPLEMENTATION OF THE 5 BAND EQUALIZER APPLICATION. THE TIME INDICATION IS RELATIVE, WITH MATLAB AS THE REFERENCE I.E. EQUAL TO 1.

		μ	σ^2	**Time**
Matlab	Double	-	-	1
SVFx1	Double	$1.586 \cdot 10^{-16}$	$1.726 \cdot 10^{-31}$	121
	24 bit fixed	$8.139 \cdot 10^{-7}$	$5.555 \cdot 10^{-14}$	121
RTL level	24 bit fixed	$8.139 \cdot 10^{-7}$	$5.555 \cdot 10^{-14}$	115

Matlab using *double*-data types.

The result of the simulation performed on the SVF processor model, implemented using double data types, matches the reference as can be seen in row one of table III. The real SVF processor model, however, is implemented using a 24 bit fixed-point data type representation. A simulation of the SVF processor model implemented using the 24 bit fixed-point representation is easily obtained based on the original SVF processor model. The simulation run is repeated and the result is shown in row two of table III. As expected, due to the smaller resolution, this result in a small mean difference compared to the double implementation.

However, more importantly, it can be seen from the table that the simulation performed on the system model implemented with the 24 bit fixed-point representation matches the results of the RTL level simulation (VHDL simulation in ModelSim [12]), shown in row three of table III, thus verifying the functionality of the SVF model.

As can be seen in the table, the time required to run the simulation in the current version of the simulator used in this method, is comparable to the time used in the RTL level simulation of the same platform. However, the time needed for constructing a platform model in the current method, compared to writing an RTL level description of the same platform, is much shorter and more straightforward. Furthermore, in this version of the simulator, the focus has been on a proof-of-concept. Thus, it is expected that the performance of the simulator can be greatly increased, and so, reduce the runtime requirements of the simulator.

V. FUTURE WORK

The current execution semantics only support the execution of models modelling synchronous hardware consisting of a single clock domain. In real world applications it is rarely the case that MPSoC systems are implemented using a single clock domain only, thus, a mandatory extension of the presented method is to enable support for modelling multiple clock domains. It should be noted that for systems consisting of multiple clock domains, asynchronous and even purely combinatorial components can be modelled if the execution semantics of the models are modified. The discussion of extending the execution semantics, however, lies outside the scope of this paper and, therefore, it should only be noted that the modelling of multiple clock domains can be done by partitioning the components of a platform model into groups and letting the maximum step occur within each group. Special care must be taken in controlling when each group is allowed to let their maximum step occur in order to accommodate the different clock frequencies modelled.

VI. CONCLUSION

This paper has presented a serviced based estimation method for MPSoC performance modelling, based on system level simulations using modified HCPNs as the model of computation. The method allows MPSoC designers to perform design space exploration of complete systems consisting of software and hardware at a very early stage in the design phase.

The method provides means for a flexible construction of models, including the modelling of components at different levels of abstraction, utilizing the composition possibilities of HCPNs. At the same time, the level of accuracy which can be obtained is very high, ranging from execution time analysis including modelling of data dependencies and resource occupancies to actual data values and word sizes as exemplified in section IV.

VII. ACKNOWLEDGEMENTS

This work was supported in part by DaNES (Danish National Advanced Technology Foundation) and ARTIST2 (IST-004527).

REFERENCES

[1] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Springer-Verlag, 1992.

[2] J. Grode and J. Madsen, "A unified component modeling approach for performance estimation in hardware/software codesign," vol. 1. IEEE, 1998, pp. 65–69.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52+4, 2003.

[4] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, "An integrated open framework for heterogeneous MPSoC design space exploration," *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, pp. 1–6, 2006.

[5] J. Ou and V. K. Prasanna, "Design space exploration using arithmetic-level hardware–software cosimulation for configurable multiprocessor platforms," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, p. 355, 2006.

[6] A. Pimentel, L. Hertzbetger, P. Lieverse, P. van der Wolf, and E. Deprettere, "Exploring embedded-systems architectures with Artemis," *Computer*, vol. 34, no. 11, pp. 57–63, 2001.

[7] Y. Yi, D. Kim, and S. Ha, "Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 12, pp. 2186–2200, 2007.

[8] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006.

[9] M. Reshadi and N. Dutt, "Generic pipelined processor modeling and high performance cycle-accurate simulator generation," *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 786–791, 2005.

[10] H. Chamberlin, *Musical Applications of Microprocessors*. Indianapolis, IN, USA: Sams, 1980.

[11] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[12] "Modelsim 6.3," http://www.model.com.